Repairing Syntax Errors in LR Parsers

RAFAEL CORCHUELO, JOSÉ A. PÉREZ, ANTONIO RUIZ, and MIGUEL TORO
Universidad de Sevilla

This article reports on an error-repair algorithm for LR parsers. It locally inserts, deletes or shifts symbols at the positions where errors are detected, thus modifying the right context in order to resume parsing on a valid piece of input. This method improves on others in that it does not require the user to provide additional information about the repair process, it does not require precalculation of auxiliary tables, and it can be easily integrated into existing LR parser generators. A Yacc-based implementation is presented along with some experimental results and comparisons with other well-known methods.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*parsing*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Automatic method, LR parsing, syntactic error repair

1. INTRODUCTION

The development of a formal theory on context-free grammars led to several efficient techniques for parsing programming languages. However, syntactic errors are frequent in practice [Ripley and Druseikis 1978], so it is desirable to enhance them with error-handling methods that allow parsing to continue when errors are detected.

Error recovery is a usual error-handling method that consists of isolating errors and changing the parse stack and the input string so that parsing can continue on some subsequent valid piece of input. Several authors have reported on such methods [Anderson et al. 1983; Sippu and Soisalon-Soininen 1982; Grosch 1990], but the one by Aho and Ullman [1972] is one of the most popular and it is available in parser generators in widespread use such as Yacc [Johnson and Sethi 1990] or Bison [Donnelly and Stallman 1990]. These techniques are efficient enough to be used in real-world compilers, but they do not attempt to repair errors and often require the user to provide additional information to

This work was supported by the Spanish Interministerial Commission on Science and Technology under grant TIC-2000-1106-C02-01.
Authors' address: Dep. de Lenguajes y Sistemas Informáticos, Escuela Técnica Superior de Ingeniería Informática, Avda. de la Reina Mercedes s/n, Sevilla E-41012, Spain; email: {corchu;jperez;aruiz;mtoro}@lsi.us.es.

deal with them, for example, error productions. Unfortunately, providing such information is difficult and the results are not as good as they should be in general.

This motivated several researchers to work on error-repair methods that are more powerful because they attempt to recover by repairing the input string so that it becomes syntactically valid [Pennello and DeRemer 1978; Röhrich 1980; Burke and Fisher 1987; Fischer and Mauney 1992; Dain 1994]. These techniques are more difficult to implement than simple recovery because they may require modification of a part of the input already parsed or the remaining input string; some such techniques require precalculation of auxiliary tables that are used to decide which kind of repair should be used when an error is detected, which also requires modification and adaptation of existing parser generators to calculate them.

This article presents an error-repair algorithm that can be used in LR parsers. Its main features are that it does not require the user to provide any information, although it allows for tuning with respect to particular languages through the setting of repair costs, it does not require precalculation of auxiliary tables, and it does not require modification of existing LR parser generators. The rest of the article is organised as follows: Section 2 presents a short introduction to LR parsing; Section 3 presents the algorithm, an example and several improvements that turn it into a practical tool; Section 4 reports on a Yacc-based implementation; a comparison with other well-known techniques is presented in Section 5; finally, Section 6 presents our main conclusions and future work.

2. LR PARSING

In this section, we review some concepts about grammars and LR parsers we need to introduce the error-repair algorithm. Further information can be found in Aho and Ullman [1972], Aho and Johnson [1974] and Hopcroft and Ullman [1979].

A *context-free grammar*, or CFG for short, describes the syntax of a language. We denote them by means of tuples of the form (N, T, S, P) where N denotes a set of *non-terminals*, T a set of *terminals*, S the *starting symbol* of the grammar, and P a set of *context-free productions*. The set $V = N \cup T$ is usually referred to as the *vocabulary* of the grammar. For instance, the following grammar describes a very simple expression language we use in the following sections:

$$G_E = (\{E\}, \{n, +, (,)\}, E, \{E ::= n, E ::= E + n, E ::= (E)\})$$

The *LR parsing method* is a technique for deciding if an input string of terminal symbols belongs to the language a CFG describes. This method scans the input string from left to right in a bottom-up manner. It starts in a state in which no input symbol has been analysed, and attempts to get back to the starting symbol by finding a sequence of symbols that constitute the right hand side of a production. If found, it is replaced with the symbol on the left-hand side of that production; otherwise, an error is detected. For instance, the input string $(n + n)$ is accepted by means of the following replacements $(E + n) \rightsquigarrow (E) \rightsquigarrow E$.

The LR parsing method[1] is formally defined by means of a deterministic transition relation $\longrightarrow_{LR}$ on configurations of the form (S, I), where S denotes a stack and I is an input string whose last symbol is the end marker \$. S maintains several pairs of the form sq, where $s \in V \cup \{\$\}$ and q denotes the state the parser enters on scanning an input symbol.[2] $\longrightarrow_{LR}$ is a general relation that needs to be customised for a particular CFG by means of an *action function* f and a *go to function* g whose signatures follow:

$$f: Q \times T \cup \{\$\} \rightarrow W$$
$$g: Q \times N \rightarrow Q$$

Here $Q = \{q_0, q_1, \ldots, q_k\}$ denotes a set of *parsing states* (q_0 is usually referred to as the initial state), and W is a set of *parsing actions* that control whether the parser shifts the current input symbol (*shift* q), reduces the stack (*reduce* $A ::= \alpha$), accepts the input string (*accept*) or detects an error (*error*).

$\longrightarrow_{LR}$ can be defined formally by means of a transition system [Plotkin 1981]:

Shift rule (LR1). This rule formalises the meaning of a shift action. If t_1 is the current input symbol, *shift* q means that a pair of the form $t_1 q$ needs to be piled up and the current input pointer can be advanced.

$$\frac{f(q_m, t_1) = shift\ \ q}{([s_0 q_0 \cdots s_m q_m], [t_1 t_2 \cdots t_n]) \longrightarrow_{LR} ([s_0 q_0 \cdots s_m q_m t_1 q], [t_2 \cdots t_n])}$$

Reduce rule (LR2). This rule formalises the meaning of an action of the form *reduce* $A ::= \alpha$. It means that $|\alpha|$ pairs need to be popped off the stack, $|\alpha|$ being the number of symbols in α; the state q_t that then appears on top, and A are used to determine the pair we need to push in order to continue parsing, which is of the form Aq, where $q = g(q_t, A)$.

$$\frac{f(q_m, t_1) = reduce\ A ::= \alpha \wedge r = |\alpha| \wedge g(q_{m-r}, A) = q}{([s_0 q_0 \cdots s_m q_m], [t_1 t_2 \cdots t_n]) \longrightarrow_{LR} ([s_0 q_0 \cdots s_{m-r} q_{m-r} A q], [t_1 t_2 \cdots t_n])}$$

A configuration $([s_0 q_0 \cdots s_m q_m], [t_1 t_2 \cdots t_n])$ is said to be *accepting* if and only if $f(q_m, t_1) = accept$, *rejecting* if and only if $f(q_m, t_1) = error$, and *intermediate* in other cases. An input string I is said to be *accepted* if and only if, starting at configuration $([\$q_0], I)$, we can reach an accepting configuration by applying $\longrightarrow_{LR}$ repeatedly; if we can reach a rejecting configuration, it is obviously said to be *rejected*.

For instance, Table I shows the action and go to functions for grammar G_E obtained with the well-known LALR(1) method. The application of $\longrightarrow_{LR}$ to the initial configuration $([\$q_0], [(n + n)\$])$ yields the results in Table II.

[1]Without lose of generality, we present our discussion in the context of LR parsers that require one lookahead symbol. This simplifies the explanations, but can be easily extended to parsers that use more symbols.

[2]Notice that it is not strictly necessary to store symbols on the stack, but it facilitates the explanations.

Table I. Action and Go To Functions for G_E (Empty entries denote errors, *shf* denotes shift, and *red* denotes reduce.)

	n	$+$	$($	$)$	\$	E
q_0	$shf\ q_2$		$shf\ q_3$			q_1
q_1		$shf\ q_4$			$accept$	
q_2		$red\ E ::= n$		$red\ E ::= n$	$red\ E ::= n$	
q_3	$shf\ q_2$		$shf\ q_3$			q_5
q_4	$shf\ q_6$					
q_5		$shf\ q_4$		$shf\ q_7$		
q_6		$red\ E ::= E + n$		$red\ E ::= E + n$	$red\ E ::= E + n$	
q_7		$red\ E ::= (E)$		$red\ E ::= (E)$	$red\ E ::= (E)$	

Table II. Application of $\longrightarrow_{LR}$ to $(n + n)$\$

Stack	Input	Rule	Action
$\$q_0$	$(n + n)$\$	LR1	*shift* q_3
$\$q_0(q_3$	$n + n)$\$	LR1	*shift* q_2
$\$q_0(q_3 n q_2$	$+n)$\$	LR2	*reduce* $E ::= n$
$\$q_0(q_3 E q_5$	$+n)$\$	LR1	*shift* q_4
$\$q_0(q_3 E q_5 + q_4$	$n)$\$	LR1	*shift* q_6
$\$q_0(q_3 E q_5 + q_4 n q_6$	$)$\$	LR2	*reduce* $E ::= E + n$
$\$q_0(q_3 E q_5$	$)$\$	LR1	*shift* q_7
$\$q_0(q_3 E q_5)q_7$	\$	LR2	*reduce* $E ::= (E)$
$\$q_0 E q_1$	\$		*accept*

3. THE ALGORITHM

The repair algorithm works on rejecting configurations and attempts to find a repair that transforms the portion of the input string that follows the position at which an error is detected into a valid one. We consider a *repair* is a sequence of insertions, deletions or shifts (with a final insertion or deletion) such that after applying it to an input string, parsing can either continue for at least N symbols or leads to an accepting configuration. The definition has been adapted from the one by Mauney [1983] and differs in that we can incorporate parse-time input symbols into repairs and parsing can continue normally for at least N symbols after repairing an error. This is a simple mechanism that improves the quality of the repairs, as we show in Section 4.1.

We introduce the algorithm by means of the following transition rule:

Repair rule (LR3).

$$\frac{(S, I) \text{ is rejecting} \wedge R \text{ is a repair} \wedge (S, I, []) \longrightarrow^*_{ER} (S', I', R)}{(S, I) \longrightarrow_{LR} (S', I')}$$

LR3 relies on a new transition relation $\longrightarrow_{ER}$ that we call the *error-repair transition*. It works on configurations of the form (S, I, R), where R is a sequence of insertions, deletions or shifts. For instance, if $R = [ins\ t, shf, del\ ]$ is a repair, it means that we need to insert t, shift the current input symbol, and delete the next input symbol to repair the input string.

Given a configuration of the form $([s_0 q_0 \cdots s_m q_m], [t_1 t_2 \cdots t_n])$, $\longrightarrow_{ER}$ deletes t_1, inserts new symbols that are acceptable at state q_m in front of t_1 or attempts to parse the remaining input string. This way, repeated application of $\longrightarrow_{ER}$

explores all of the possible configurations we can reach by inserting, deleting or shifting symbols when an error is detected. Thus, it eventually finds a configuration (S', I', R) such that parsing can continue normally on (S', I'). This configuration exists because, in the worst case, we can delete the remaining input string and then insert suitable symbols to complete it.

We define $\longrightarrow_{ER}$ by means of three rules in which $S = [s_0 q_0 \dots s_m q_m]$, $I = [t_1 t_2 \cdots t_j \cdots t_n]$, and $R = [r_1, r_2, \dots, r_k]$ $(m, n \geq 1$, and $k \geq 0)$:

Insertion rule (ER1). $\longrightarrow_{ER}$ searches the action function for symbols $t_0 \neq \$$ such that $f(q_m, t_0) \neq error$ and creates new configurations in which t_0 is in front of t_1.

$$\frac{f(q_m, t_0) \neq error \wedge t_0 \neq \$ \wedge (S, [t_0 t_1 t_2 \cdots t_j \cdots t_n]) \longrightarrow^*_{LR} (S', I)}{(S, I, R) \longrightarrow_{ER} (S', I, [r_1, r_2, \dots, r_k, ins\ t_0])}$$

Deletion rule (ER2). It also produces a new configuration by deleting t_1.

$$\frac{n \geq 2}{(S, I, R) \longrightarrow_{ER} (S, [t_2 \cdots t_n], [r_1, r_2, \dots, r_k, del\ ])}$$

Forward move rule (ER3). If $\longrightarrow_{ER}$ is applied to an intermediate configuration, it parses the remaining input symbols, that is, applies $\longrightarrow_{LR}$, until a new error is found, an accepting configuration is reached, or N symbols have been parsed.

$$\frac{(S, I) \longrightarrow^*_{LR} (S', I') \wedge (j = N \vee 0 < j < N \wedge f(q_r, t_{j+1}) \in \{accept, error\})}{(S, I, R) \longrightarrow_{ER} (S', I', R')}$$

where $S' = [s_0 q_0 \cdots s_r q_r]$, $I' = [t_{j+1} \cdots t_n]$, $R' = [r_1, r_2, \dots, r_k, \overbrace{shf, \dots, shf}^{j}]$, $r, j \geq 1$, and $k \geq 0$.

3.1 An Example

To illustrate how the algorithm works, we confront it with $(n_1 n_2\$$. (Subscripts have been added for the sake of clarity.) Repeated application of $\longrightarrow_{LR}$ to the initial configuration $([\$q_0], [(n_1 n_2\$])$ yields $([\$q_0(q_3], [n_1 n_2\$])$, $([\$q_0(q_3 n_1 q_2], [n_2\$])$ and an error is detected immediately. A new configuration of the form $([\$q_0(q_3 n_1 q_2], [n_2\$], [])$ is then built and rules ER1 and ER2 can be applied yielding configurations

$$C_1 = ([\$q_0(q_3 E q_5 + q_4], [n_2\$], [ins\ +])$$
$$C_2 = ([\$q_0(q_3 E q_5) q_7], [n_2\$], [ins\ )])$$
$$C_3 = ([\$q_0(q_3 n_1 q_2], [\$], [del\ ])$$

Now, we can inspect configuration C_1 and apply rule ER3, which leads to the following configuration (Notice that ER1 and ER2 are also applicable. Later, we present some heuristics to speed up the search process that justify our selection.):

$$C_4 = ([\$q_0(q_3 E q_5], [\$], [ins\ +, shf\ ])$$

Here, a new error is detected at state q_5, so ER1 is now the only applicable rule and it produces two new configurations:

$$C_5 = ([\$q_0(q_3 E q_5)q_7], [\$], [ins\ +, shf, ins\ )])$$
$$C_6 = ([\$q_0(q_3 E q_5 + q_4], [\$], [ins\ +, shf, ins\ +])$$

Now, we can apply rule ER3 to C_5 so that we reach the following repair:

$$C_7 = ([\$q_0 E q_1], [\$], [ins\ +, shf, ins\ )])$$

Therefore, inserting a +, shifting n_2, and then inserting a) is a repair for the incorrect input string $(n_1 n_2\$$, and it transforms it into $(n_1 + n_2)\$$. Nevertheless, there are infinitely many other repairs. For instance, starting at configuration C_2, we can also reach a repair that transforms the erroneous input into $(n_1) + n_2\$$, starting at C_3 we can transform it into $(n_1)\$$, and starting at configuration C_6, we can transform it into $(n_1 + n_2 + \cdots + n_k)\$$ for any $k \geq 3$.

3.2 Problems and Solutions

Unfortunately, the algorithm described above suffers from several drawbacks:

(1) Several repairs might be possible because of the non-determinism embodied in the definition of $\longrightarrow_{ER}$. Deleting the remaining input and inserting adequate symbols is always possible, but it should be avoided as well as repairs that result in too many changes to the input string if others that require fewer changes are possible.

(2) $\longrightarrow_{ER}$ may generate an infinite set of configurations, so a huge search space might be unsuccessfully explored before finding a suitable repair, which results in a waste of user and computer time. Furthermore, if we do not use an adequate search procedure, it might loop while traversing an infinite branch. For instance, when state q_5 is on top of the stack, inserting a + and then an n is always possible and would produce an infinite set of configurations.

(3) There is no upper limit to the amount of insertions, deletions or shifts a configuration may need to become repaired.

Fortunately, these problems can be solved efficiently, according to the results we present in Section 4. As for the first two problems, Anderson and Backhouse [1981] proposed a method called *least-cost recovery* that associates a cost with each operation on the input string. Thus, each repair has an associated cost that can be used to select among several repairs and to prune the search space. If this method is used, then the least-cost repair is always chosen and the first repair $\longrightarrow_{ER}$ finds can be used to prune configurations with a higher cost, that is, a configuration is explored as long as its cost is smaller than that of the best configuration found so far. However, finding out these costs is a difficult task where intuition and experimentation are the most powerful tools we can use. Therefore, if no good costs are available, we can use a *simpler heuristic*: we select the configuration that results in the smallest number of changes to the input string. If there are several possibilities, we then let the user favor the one with a smaller number of insertions or

deletions, and if there are still several possibilities, one of them is selected arbitrarily.

Preventing the parser from looping while traversing an infinite branch is easy if we do it in a breadth-first manner. Nevertheless, we still have not introduced an upper limit to the amount of insertions, deletions or shifts, and some errors could take arbitrarily long to become repaired. For instance, repairing an input string of the form $((\cdots((n\$$ takes longer as the number of mismatched parentheses increases. In practice, limiting the repair process to a portion of N_t symbols, the number of insertions to N_i, and the number of deletions to N_d (where N_t, N_i and N_d are constants), proved to work well and produced good results, as we show in Sections 4.1 and 4.2. However, if we introduce these limits, the algorithm might not produce a result because there might not exist a repair of the input involving 4 insertions and 3 deletions in a fixed-sized region of 10 symbols, for instance. In those cases, reverting to an efficient secondary recovery mechanism is a good idea, and we have selected *panic mode* [Holub 1990]. This method consists of scanning down the stack until a state accepting the erroneous symbol is found or the stack is emptied. In the former case, parsing resumes, and in the latter the stack is restored, the erroneous symbol is deleted and the procedure applied again to the resulting configuration. If the input string is emptied, then the procedure fails to recover.

Finally, another improvement that significantly reduces the size of the search space consists of not examining configurations in which the last operation is a deletion for further insertions. For instance, if state q_5 is on top of the stack and n is the current input symbol, deleting n and then inserting $+$ or) results in the same configurations as inserting $+$ or) and then deleting n. This way, we avoid duplication of effort in many usual cases.

4. THE IMPLEMENTATION

We have incorporated the repair algorithm into PC-Yacc [Lane 1989], a widely available, efficient implementation of Johnson's Yacc. It was not modified, but the driver we need to interpret the LALR(1) tables it produces. We maintain a queue of configurations (S, I, R) and explore them sequentially. New configurations are added at the end of the queue, thus simulating a breadth-first traversal. As for the parameters of the algorithm, we have found through experimentation that setting $N = 3$, $N_t = 10$, $N_i = 4$ and $N_d = 3$ produces good repairs.

The implementation was complicated by the fact that PC-YACC uses default reductions and row compression [Aho et al. 1986] to compact the parsing tables it produces. The main consequence of default reductions is that error detection may be delayed because of an undesirable default reduction on scanning an erroneous symbol. This is a well-known issue that also arises in SLR and LALR parsers. Burke and Fisher [1987], for instance, used a technique called deferred parsing that solves it, but slows down parsing by about 10%. We have implemented a simpler solution: consider a configuration of the form $([s_0 q_0 \cdots s_m q_m], [t_1 t_2 \cdots t_n])$ so that $f(q_m, t_1) = \textit{reduce } A ::= \alpha$; to determine if t_1 is a valid symbol we do not actually need to do the reduction, we only need

Table III. Quality of Repairs

Authors	Excellent	Good	Poor	Unrepaired	Acceptable
Pennello & DeRemer	42.0%	28.0%	12.0%	18.0%	70.0%
Dain ($\sigma = 5$)	67.0%	14.0%	18.2%	0.8%	81.0%
Corchuelo & al. (without costs)	57.2%	25.2%	16.8%	0.8%	82.4%
Corchuelo & al. (with costs)	62.3%	24.3%	12.6%	0.8%	86.6%
Fischer & LeBlanc	75.0%	21.0%	4.0%	0.0%	96.0%
Burke & Fisher	77.6%	20.0%	2.4%	0.0%	97.6%

to consult state $q_{m-|\alpha|}$. If $f(q_{m-|\alpha|}, t_1)$ is not a reduction, then we can decide whether t_1 is valid or not; otherwise, we can repeat the same procedure until the chain of reductions finishes. This method can be implemented efficiently, and the experiments we conducted showed that the slowdown it produces is negligible (less than 0.1%).

4.1 Repairs

Our performance evaluation focused on Pascal owing to two reasons: (i) the language we use to teach our students compilers is a Pascal clone, so we have many genuine syntactically-invalid programs they submit for compilation; (ii) many authors have used the well-known collection of erroneous Pascal programs by Ripley and Druseikis [1978] to evaluate the quality of their algorithms, so comparisons with them is possible for Pascal.

The quality of a repair is usually measured using the categories proposed by Pennello and DeRemer [1978]. Table III presents a summary that compares our algorithm with other authors' proposals that are examined in Section 5. Notice that about 87% of the repairs were acceptable (excellent or good), with only 12.6% judged poor using insertion/deletion costs; the quality using our simpler heuristic still rates comparably with about 82% of acceptable repairs.

The parser was not able to repair the errors in the input, that is, it resorted to panic mode, only in the following situation:

```
FUNCTION SEARCH(X: ORDER;): BOOLEAN;
   VAR Q: INTEGER;
BEGIN X := 1 END;
```

When the error is detected at the closing parenthesis, a block of formal parameters has just been reduced and the semicolon after it indicates that a new block should begin there. The algorithm thus deletes the closing parenthesis and inserts a new identifier, which transforms the input into:

```
FUNCTION SEARCH(X: ORDER; NEW_ID: BOOLEAN;
   VAR Q: INTEGER;
BEGIN X := 1 END;
```

Now, a new error is detected at the key word **BEGIN**. The algorithm might have inserted a string to complete the header, but this would have required the insertion of more than 4 symbols. The suite contains 197 errors, and we think that resorting to panic mode in one situation is quite an acceptable ratio.

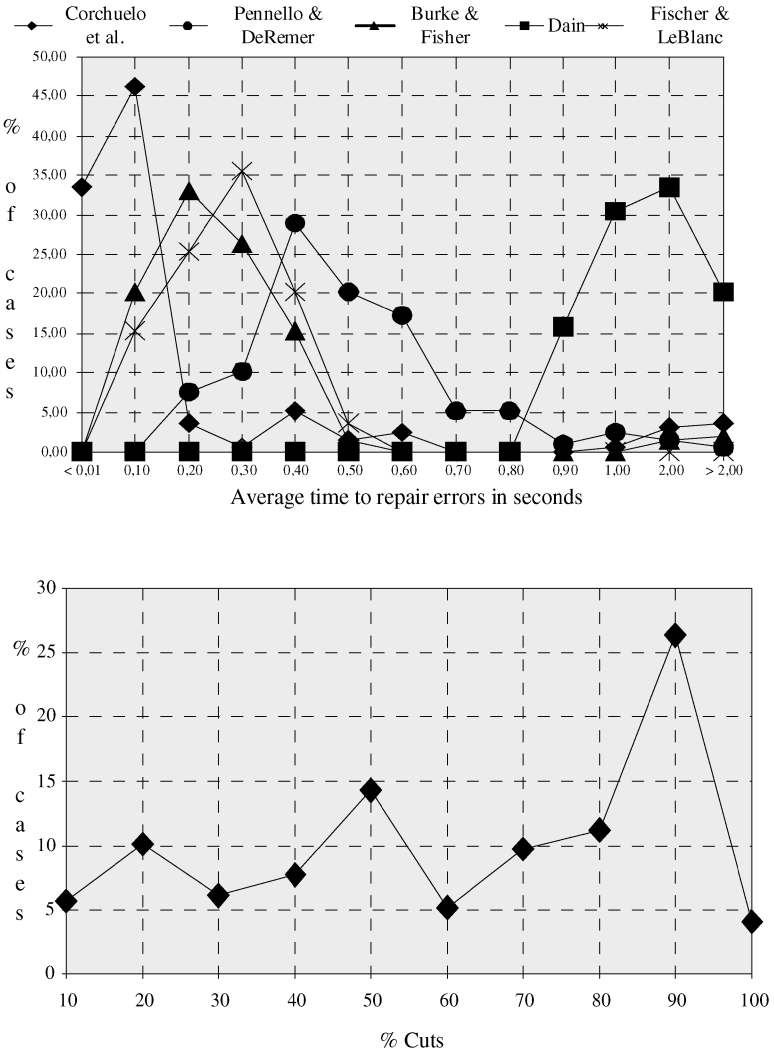

Fig. 1. Time to repair individual errors, and performance of the prune heuristic.

4.2 Efficiency

The algorithm is also efficient, in both space and time. We tested it on a 200 Mhz Pentium computer in which the amount of available data memory was limited to 64 KBytes. The parser never ran out of memory, thus proving that it does not produce a significant memory overhead. Figure 1 summarises the time our algorithm took to repair errors and compares it with other proposals. As shown, more than 80% of the errors were repaired in less than 0.1 seconds, whereas other algorithms that produce better quality repairs obviously take longer to repair. Our pruning techniques were also proved very adequate, as shown in the same figure, because they helped us cut more than 70% of the search space in 50% of the cases.

5. RELATED WORK

We first present some state-of-the-art continuation-based methods. Continuations were defined by Röhrich [1980] as strings that can be inserted at the positions where errors occur and allow the parser to be restarted without further errors. He proposed a method that consists of generating a continuation and deleting symbols from the remaining input until an anchor symbol contained in the continuation is found; the appropriate prefix of the continuation is then inserted and parsing resumes on the anchor. Dion [1982] improved Röhrich's idea and developed a method that generates a least-cost continuation, deletes some input symbols and inserts a prefix of the continuation to resume parsing. It is analogous to the LL(1) algorithm by Fischer et al. [1980] and uses user-defined insert/delete costs to precalculate two tables called $S[A]$ and $E[A, t]$ ($A \in N, t \in T$); $S[A]$ gives the lowest cost string derivable from A, and $E[A, t]$ the lowest cost prefix that allows t to be derived from A. Unfortunately, the size of these tables is usually comparable to the size of the parsing tables for the grammar under consideration. Fischer and LeBlanc [1988] improved Dion's algorithm using a scheme that allows repairs as effective as Dion's to be issued using smaller tables. Dain [1994] developed another continuation-based method that consists of generating the set of prefixes of continuation strings of length σ and replacing a prefix of the remaining input string with the one whose distance is minimum according to the similarity criterion by Wagner and Fischer [1974].

Our method is continuation-based, but it does not generate continuations without taking into account the remaining input, as is the case with the above-mentioned methods. For instance, Röhrich's method generates only an arbitrary continuation and, thus, tends to produce poor quality repairs. Both Dion's and Fischer and LeBlanc's methods select the least-cost continuation according to user-defined insertion/deletion costs, but the problem is that some of its symbols might be already part of the input string, which implies we are deleting and inserting them and increasing the cost of the repair. Our algorithm, instead, takes input symbols into account and they are not considered when we calculate the cost of a repair. Dain's method selects the prefix of the continuation that best matches the input string, thus taking input symbols into account. However, the problem is that this method needs to generate the whole set of prefixes of continuation strings by exploring the valid transitions at the state where an error is detected, which implies the method requires more and more memory and time as σ increases. Unfortunately, the whole search space has to be explored and no validation is performed, which results in poor repairs if several errors are close.

Furthermore, the only continuation-based method that can be easily incorporated into existing LR parser generators is Dain's because it requires modification only of the driver used to interpret parsing tables. The other continuation-based methods require significant modification to produce additional information. Fischer and LeBlanc's method even requires using a special LR closure algorithm in order to generate least-cost continuations and the resulting parsing tables may be slightly larger than usual.

Pennello and DeRemer [1978] and Burke and Fisher [1987] presented two proposals that deserve special attention. The former is an implementation of the general idea by Graham and Rhodes [1975] in the context of LR parsers that improves the proposal by Mickunas and Modry [1978]. It consists of two phases: forward move, whose goal is to gather right context, and repair, in which a number of edit operations are performed on the input string. The first phase is carried out by means of a forward move automaton (FMA) that allows parsing the remaining input until a new error is found or a reduction across the erroneous symbol is attempted; if the FMA halts before reaching the end of the input string, it begins parsing again from the symbol following the last symbol it was able to shift (seven times at least). The FMA performs all possible parses in parallel, so the forward move produces a sequence of configurations that represent all partial parses of the remaining input. The repair phase tries to link the sequence of configurations the FMA produces with the configuration in which the error was detected by deleting the erroneous symbol, replacing it, or inserting a new one; if none succeeds, the stack is then backed up and the procedure is applied again. Furthermore, before trying insertions, the algorithm tries to attach the erroneous symbol to the right context. If it fails, the proposal lacks a systematic method for dealing with close errors in the right context; however, the authors suggest using a technique called stack forcing, which analyses the stack and attempts to insert symbols in it so that it can be linked to the configurations the FMA produced.

The forward move phase bears some resemblance to our forward move rule, but there are two important differences: (i) the FMA does not attempt to correct new errors, whereas our technique is applied recursively in order to repair nearby errors; (ii) the FMA does not take the left context into account, whereas our technique can perform reductions across the error detection point in an attempt to correct it. Furthermore, the FMA requires about 20–50% of the space needed to store the parsing tables for the grammar under consideration, and it cannot be generated by existing parser generators unless we modify them. The need to parse the remaining input (or seven fragments at least) also introduces additional penalty and makes the time an error needs to become repaired dependent on the length of the input string, which is undesirable. The quality of the repairs is worst among the algorithms we have compared in Table III. The number of poor repairs is similar to our proposal, but the number of unrepaired errors is significantly greater. Furthermore, stack forcing may require inserting a non-terminal symbol into the stack, which is not desirable because it can invalidate semantic actions. The error messages the algorithm produces are apt to be confusing because they include non-terminal symbols that usually do not make sense to the final user.

The technique by Burke and Fisher [1987] consists of three phases: it first attempts to repair the error by inserting or deleting a single symbol, substituting the erroneous symbol for another, or merging the next two symbols into one; if this simple repair fails, the method tries then to use scope recovery, which attempts to close one or more open scopes; if it fails, then some text surrounding the erroneous symbol is deleted. The main problem this technique faces is that in LL, LALR, SLR or LR implementations that perform default

reductions, an erroneous symbol may induce several undesirable reduce actions that inevitably transform the parse stack, which frequently leads to poor repairs. Burke and Fisher [1987] used a technique called deferred parsing that may be viewed as double parsing: one parser goes ahead as much as possible, whereas the other is k steps behind, so that unparsing to the a state k steps before is very easy. The algorithm by Burke and Fisher produces, to the best of our knowledge, the best quality repairs; its main problem is that deferred parsing slows down parsing of valid inputs for any $k > 0$, which is not desirable in general. The authors report that this penalty is about 10% in both space and time, and that setting $k > 1$ does not result in significantly better repairs for Pascal. To an extent, our implementation simulates deferred parsing with $k = 1$ because we examine the stack to check if the current input symbol is valid before performing a reduction, but this does not entail neither significant time nor space overhead.

6. CONCLUSIONS AND FUTURE WORK

A repair algorithm for LR parsers has been presented and defined formally. It improves on others because it does not require the user to provide additional information, it does not require precalculation of auxiliary tables, it uses input symbols to complete repairs, and it can be incorporated into existing LR parser generators by only modifying the drivers that are provided to interpret parsing tables. From this point of view, it does not only compare to Dain's proposal, but improves it in both quality and efficiency.

The experimental results show that it is efficient enough to be used in practical applications in which it is necessary to spend little time or memory at repairing. The average time per error is the smallest among the algorithms we examined, but the quality of the repairs still rates comparably with state-of-the-art methods.

In the future, we are going to incorporate it into Bison, which stores parsing tables in a format that is not compatible with PC-Yacc. Much attention is also going to be paid to the application to LL and LR hard-coded parsers [Bhamidipaty and Proebsting 1998].

ACKNOWLEDGMENTS

The authors wish to thank their referees for their careful reading and thoughtful suggestions, and Dr. David Ripley for sharing his collection of erroneous Pascal programs with us.

REFERENCES

AHO, A. AND JOHNSON, S. 1974. LR parsing. *ACM Comput. Surv. 6*, 2 (June), 99–124.

AHO, A., SETHI, V., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Menlo Park, California.

AHO, A. AND ULLMAN, J. 1972. *The Theory of Parsing, Translation and Compiling*. Prentice Hall, Englewood Cliffs, New Jersey.

ANDERSON, S. AND BACKHOUSE, R. 1981. Locally least-cost error recovery in Early's algorithm. *ACM Trans. Program. Lang. Syst. 3*, 3 (July), 318–347.

ANDERSON, S., BACKHOUSE, R., BUGGE, E., AND STIRLING, C. 1983. An assessment of locally least-cost error recovery. *Comput. J. 26*, 1 (Feb.), 15–24.

BHAMIDIPATY, A. AND PROEBSTING, T. 1998. Very fast Yacc–compatible parsers (For very little Effort). *Softw. Pract. Exper. 28*, 2 (Feb.), 181–190.

BURKE, M. AND FISHER, G. 1987. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst. 9*, 2 (Apr.), 164–197.

DAIN, J. 1994. A practical minimum distance method for syntax error handling. *Comput. Lang. 20*, 4 (Nov.), 239–252.

DION, B. 1982. Locally least–cost error correctors for context–free and context–sensitive parsers. Ph.D. thesis, University of Wisconsin at Madison.

DONNELLY, C. AND STALLMAN, R. 1990. *Bison: the Yacc–compatible parser generator*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 0219, USA.

FISCHER, C. AND LEBLANC, R. 1988. *Crafting a Compiler*. Benjamin/Cummings Series in Computer Science. The Benjamin–Cummings Publishing Company, Menlo Park, California.

FISCHER, C. AND MAUNEY, J. 1992. A simple, fast, and effective LL(1) error repair algorithm. *Acta Inf. 29*, 2, 19–120.

FISCHER, C., MILTON, D., AND QUIRING, S. 1980. Efficient LL(1) error correction and recovery using only insertions. *Acta Inf. 13*, 2, 141–154.

GRAHAM, S. AND RHODES, S. 1975. Practical syntactic error recovery. *Commun. ACM 18*, 11 (Nov.), 639–650.

GROSCH, J. 1990. Efficient and comfortable error recovery in recursive descent parsers. *Struct. Program. 11*, 3, 19–140.

HOLUB, A. 1990. *Compiler Design in C*. Prentice Hall, Englewood Cliffs, New Jersey.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts.

JOHNSON, S. AND SETHI, R. 1990. Yacc: A parser generator. In *Unix Research System*. Vol. II. Saunders College Publishing, Philadelphia, Pennsylvania, 347–374.

LANE, A. 1989. Generating parsers with PC–Yacc. *Dr. Dobb's Journal of Software Tools 14*, 6 (June), 76–77, 79, 81, 110–112.

MAUNEY, J. 1983. Least–cost syntactic error repair using extended right context. Ph.D. thesis, University of Wisconsin at Madison.

MICKUNAS, M. AND MODRY, J. 1978. Automatic error recovery for LR parsers. *Commun. ACM 21*, 6 (June), 459–465.

PENNELLO, T. AND DEREMER, F. 1978. A forward move algorithm for LR error recovery. In *Conference Record of The 5th Annual ACM Symposium of Principles of Programming Languages*. ACM, Tucson, Arizona, 241–254.

PLOTKIN, G. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN–19, Computer Science Department, Aarthus University.

RIPLEY, G. AND DRUSEIKIS, F. 1978. A statistical analysis of syntax errors. *Comput. Lang. 3*, 4, 227–240.

RÖHRICH, J. 1980. Methods for the automatic construction of error correcting parsers. *Acta Inf. 13*, 2 (Feb.), 115–139.

SIPPU, S. AND SOISALON-SOININEN, E. 1982. Practical error recovery in LR parsing. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*. ACM, Albuquerque, New Mexico, 177–184.

WAGNER, R. AND FISCHER, M. 1974. The string-to-string correction problem. *J. ACM 21*, 1 (Jan.), 168–173.

Received July 2000; revised November 2001; accepted May 2002